新連載

SystemVerilogシミュレーション 活用チュートリアル 01

第1回 基本的なシミュレーションの手順

森田栄一

デバイスの記事　関連データ

本連載では，市販シミュレータの評価版を操作しながら，SystemVerilogを利用した検証手法について学習していく．サンプル設計は，プロセッサ，メモリ，キャッシュから構成されるシステムである．シミュレータとして，米国Mentor Graphics社の「Questa」を使用する．ここで使用するシミュレータの評価版とサンプル設計は，本誌付属のCD-ROMに収録されている．（編集部）

ここ1，2年で，SystemVerilog，PSL，アサーション，テストベンチ自動化などのキーワードをよく耳にするようになりました．このようなキーワードと「検証」をテーマにしたセミナなども頻繁に開催されており，多くの設計・検証エンジニアが新しい検証手法に目を向けています．

しかしここで言う「新しい検証手法」は，じつはとくに斬新で難しい概念というわけではありません．テストベンチの自動化やアサーションといった手法は，その標準的な手法が確立されていなかっただけで，コンセプトは以前からありました．ただ標準的な方法がなかったため，対応する検証ソフトウェアも少なく，プロジェクトごとに独自のツールを作成したり，C言語によるテストベンチを書いたりと，手間のかかる方法で実現していたのです．

これらの方法については，手法そのものの信頼性についても検証が必要となり，設計データにバグがあるのか検証方法がまちがっているのかについての切り分けが困難になるケースが発生します．SystemVerilogは，このような検証手法を共通化するため，Verilog HDLをベースに開発された設計・検証言語なのです．

本連載では，SystemVerilogと検証ツール（シミュレータ）を用いて，テストベンチの自動化とアサーションによるデバッグ手法について体験していきます．第1回は検証ツールの基本的な使用方法について解説します．その後，テストベンチの自動化（第2回），アサーション・ベース検証手法（第3回）と話を進めていきます．

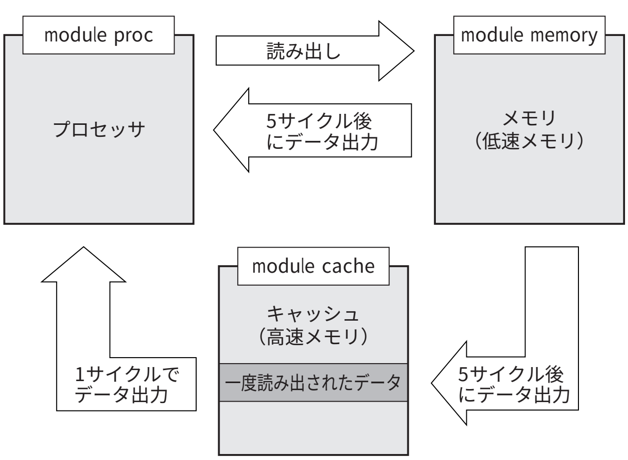


図1　サンプル設計のイメージ

メモリからの読み出しでは，通常，プロセッサが要求を出してからメモリの動作時間分の遅延（レイテンシ）後にしか完了できない．メモリから読み出す際に同時にキャッシュ・メモリにデータを保存し，2回目以降はキャッシュ・メモリからデータを読み出すことで，処理を高速化できる．

● 使用するツールとサンプル設計

本連載では，検証ツールに米国Mentor Graphics社のHDLシミュレータ「Questa」を用います．本ツールは，VHDLとVerilog HDLの論理シミュレータをベースに，

KeyWord　SystemVerilog，Questa，テストベンチ，検証，キャッシュ，メモリ，デバッグ，疑似LRUアルゴリズム，ライト・スルー方式，インクリメンタル・コンパイル

SystemVerilogなどのシミュレーション機能を搭載した検証ソフトウェアです．評価版が本誌の付属CD-ROMに収録されています．インストールの方法を記載したPDFファイルもいっしょに収録されているので，こちらを参考に，ソフトウェアのセットアップを完了してください．

チュートリアル用のサンプル・データも同じようにCD-ROMに収録されています．ディレクトリごとハード・ディスクにコピーして使用します．以下ではデータを「C:\example」にコピーした，という前提で説明を進めます．皆さんもとくに不都合がないかぎり，同名のディレクトリにコピーしてください．

サンプル設計は次の四つのファイルを含んでいます．

- top.sv　：テスト環境のトップレベル（最上位階層）
- cache.sv　：キャッシュとキャッシュ・コントローラ
- proc.sv　：プロセッサ・モデル
- memory.sv：メモリ・モデル

図1が設計全体のイメージです．トップレベル以下に三つのモジュール`proc`，`cache`，`memory`があります．本連載中では，それぞれのモジュールをプロセッサ，キャッシュ，メモリと表記します．この中で，キャッシュが検証の対象（DUT：design under test）になります．

プロセッサとメモリは，それそのものは検証の対象ではなく，検証の環境を構成するキャッシュの周辺モジュールです．このようなモジュールを「シミュレーション・モデル」と呼びます．プロセッサとメモリは，メモリのリード（読み出し）とライト（書き込み）のやりとりだけの単純な動作しか行いません．このプロセッサとメモリの間にキャッシュを挿入することで，検証環境を構成します．

● 検証対象はキャッシュ・メモリ

キャッシュ（キャッシュ・メモリ）とは，高速に動作するプロセッサなどのデバイスと，動作の遅いメモリの間に挿入する高速なメモリのことです．

図1のように，メモリがデータを出力するのに5サイクルかかるとします．このシステムに1サイクルでデータの読み出しが可能なキャッシュを挿入します．プロセッサからリード命令が発行されると，メモリからプロセッサにデータを読み込むのと同時にキャッシュの該当アドレスにもデータを読み込みます．プロセッサが再度同じアドレスのメモリ・データを読み込む際には，キャッシュからデータを読み込むことで高速にデータにアクセスすることが可能です．このキャッシュは高速ですが，高価なことやチップ上に集積される際にチップ面積の制約を受けるなどの理由により，メモリと比べると小容量しか実装されません（もちろんメモリと同容量を準備できれば，メモリそのものが不要になる）．

この容量の小さいキャッシュでは，メモリの全内容を保持することはできません．そこでキャッシュを制御するコントローラで使用頻度の高いデータを選択し，保持しておく必要があります．キャッシュ内を検索して該当のメモリ・アドレスのデータが保存されていた場合は「キャッシュ・ヒット」，キャッシュ上に存在しない場合は「キャッシュ・ミス」と呼びます．キャッシュのヒット率は，システムの性能を決める大きな要因となります．

● 3段階でシミュレーションを実行

今回は，三つのステップでシミュレーションを行います．第1ステップのシミュレーションでは，キャッシュを使用せずにプロセッサとメモリのみのデータのやりとりを観測します．これにより，リード/ライト・サイクルにおける各信号の動作とタイミング，およびメモリ・デバッガの使用方法を確認します．第2ステップではキャッシュを挿入して，第1ステップとまったく同一の信号パターン（シミュレーション・パターン）を与えることで，キャッシュの効果を確認します．第3ステップでは信号パターンを編集してより多くの動作パターンを確認します．

シミュレーションと検証のフローは**図2**のようになります．このフローは，ほとんどのシミュレータ・ソフトウェアで共通ですが，実行するコマンドなどはソフトウェアによって異なります．

本チュートリアルでは，ソース・コードは準備されてい

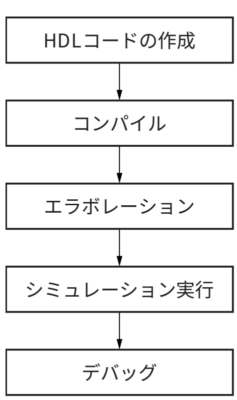


図2
シミュレーションと検証のフロー
ここでは，シミュレーションと検証に焦点をあてている．さらにこの後，論理合成などのステップに進むことになる．

るのでとくに記述を行う必要はありません（第3ステップでは若干の編集を行う）．次のステップの「コンパイル」は，HDLソース・コードが文法どおりに記述されているかどうかをチェック（パーサと呼ばれるプログラムが実行する）し，シミュレータで実行可能なバイナリ・イメージを生成します．「エラボレーション」とは，コンパイルされた設計データを組み上げて，一つのシミュレーション可能なデータを構築することです．エラボレーションは，シミュレータの用語では「ローディング」という言いかたをすることもあります．

「シミュレーションの実行」と「デバッグ」では，設計データに対して入力信号を与えて時間を進めたり，止めたりしながら出力結果をモニタしていきます．デバッグの結果，問題が発見された場合には，HDLコードの修正を行ってシミュレーション・フローを再実行します．

● シミュレータを起動する

デスクトップ上のアイコンをクリックして，Questaを起動します．最初の起動時には，ファイルの拡張子を登録するダイアログ・ボックスが表示される場合がありますが，本ツールを関連付けしたい場合を除いて，［いいえ］ボタンを選択してください．また，本ツールのWelcomeウィンドウが表示された場合は，「Don t show this dialog again」を選択して，［Close］ボタンをクリックしてください．

図3が，起動時に表示されるメイン・ウィンドウです．最初はWorkspaceとTranscriptが表示されています．Workspaceには，設定されているライブラリの情報や，設計の階層ツリーなどが表示されます．表示する情報が増えるごとにタブが追加されていきます．Transcriptはコマンド・ライン・インターフェースで，本チュートリアルの中では「コマンド・ライン」と表記します．

メイン・ウィンドウのメニューから「File」→「Change Directory」を選択し，**図4**のダイアログ・ボックスを表示します．ここから設計データをコピーしたディレクトリへ移動します．ブラウザで「C:\example」を選択して，［OK］ボタンをクリックします．

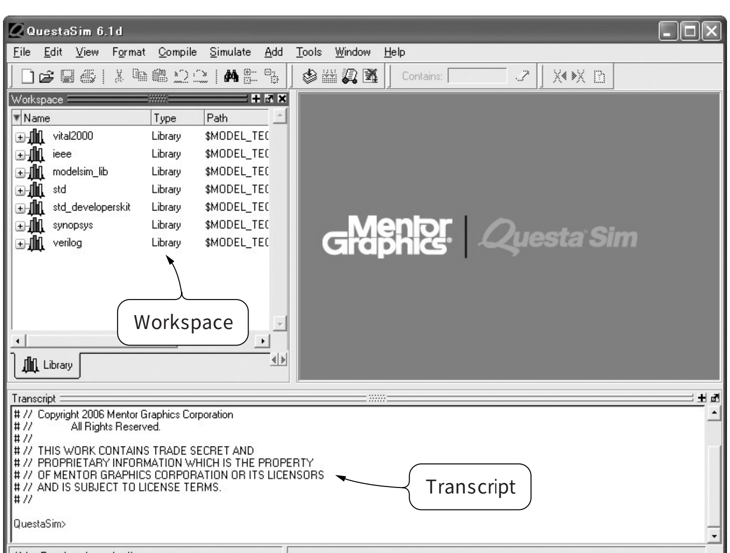
● ライブラリを準備してコンパイル

ソース・コードをコンパイルする前に，コンパイルしたデータを格納するためのライブラリを作成します．この「ライブラリ」という呼びかたや，コンパイル・データの格納の方法は，シミュレータによって若干異なります．ここでの手順は，本ツールに特有の方法となります．コマンド・ラインに，次のコマンドを入力します．

```
QuestaSim> vlib work
```

これで，workという名称のライブラリ・ディレクトリが作成されます．Windowsエクスプローラで確認すると，workという名称のディレクトリが見つかります．引き続きコンパイル作業を行います．次のコマンドを入力します．

```
QuestaSim> vlog *.sv
```



◀ **図3 メイン・ウィンドウ**
Workspaceには設計データの情報が表示される．Transcriptはコマンド・ライン・インターフェース．

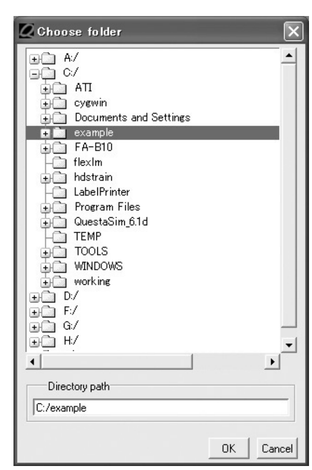


図4　ディレクトリの移動
ディレクトリ・ブラウザから作業用のディレクトリに移動した後，シミュレーション・フローを実行する．

コンパイラの引き数には，ワイルド・カード(*)を使用できます．コンパイルを実行すると，次のようなメッセージが表示されて，topとtop_pmという二つのトップが認識されていることがわかります．

```
# Top level modules:
# top
# top_pm
```

このtopがキャッシュのテスト環境です．top_pmは，テスト環境からキャッシュを除外して，プロセッサとメモリのみで構成されています．

● 設計データをローディングする

第1ステップのシミュレーションでは，top_pmをローディングします．次のコマンドを入力します．

```
QuestaSim> vsim -wlf pm.wlf top_pm
```

「-wlf pm.wlf」というオプションを用いてシミュレーション結果のログ・ファイル名を指定しています．無指定の場合はvsim.wlfという名称で保存されますが，シミュレーション起動ごとに上書きされてしまうので，後で利用するために名まえを付けておきます．設計データがローディングされて，**図5**のようなGUIが表示されます．また，コマンド・プロンプトが「QuestaSim>」から「VSIM>」に変わって，シミュレーションのセッションが起動されたことを表します．

Workspaceには設計の階層構造が表示されています．ここでundockアイコン をクリックしてWorkspaceをアンドックします(**図6**)．「アンドック」とは，メイン・ウィンドウのフレームからサブウィンドウを独立表示させることで，サブウィンドウのdockアイコン をクリックすることによりメイン・ウィンドウ内に再ドックできます．また，ドックされた状態ではサブウィンドウのフレームをドラッグすることでウィンドウの表示位置や大きさを変更することが可能です．

アンドックしたWorkspace上で各階層の「+」マークをクリックすると，階層ツリーを展開することができます．ここでの表記方法は， マークは階層やタスクに， マークはVerilog HDLプロセス(assign文，initial文，always文など)になります．モジュールtop_pm以下に，モジュールprocとmemoryがpとmという名まえでそれぞれインスタンスされていることがわかります．また，proc内には，readとwriteというメモリ・アクセスのためのタスクが記述されていることを確認できます．階層構造を確認したあと，Workspaceを再ドックします．

● 波形を見ながらデバッグ

シミュレーションのデバッグのもっとも基本的な方法は



図5 設計データのローディング
Workspaceにツリーが表示される．Workspaceの階層を選択することで，Objectsウィンドウの表示は自動的に更新される．

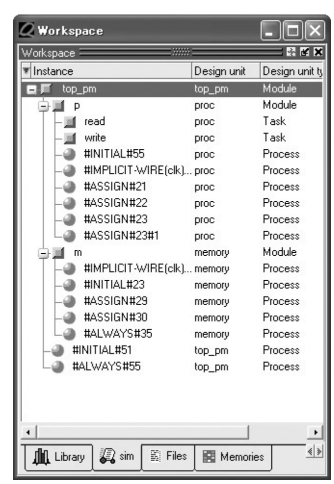


図6 アンドックしたWorkspace
メイン・ウィンドウ内のサブウィンドウは，アンドックできる．再び，Dockボタンをクリックすると，メイン・ウィンドウにドックされる．

「波形によるデバッグ」です．HDLを用いたディジタル・シミュレーションでは，‘0’と‘1’で表現される2進数の値を観測することで動作を確認します．制御信号などはほとんどが1ビットで構成されるので，‘0’と‘1’の値の状態を確認します．また，メモリの値など，一定のビット幅を持つ信号は16進表記や10進表記などを用いて観測します．

実際に波形を表示するためには，最初に観測する信号をリストアップしておく必要があります．このシミュレーションでは，最上位階層のプロセッサとメモリの間を接続する信号を観測します．

次のコマンドを入力して観測信号を登録します．

```
VSIM> add wave /*top_pm/*
```

波形モニタのWaveウィンドウが表示されます．引き続き，次のコマンドを用いてWaveウィンドウをアンドックします．

```
VSIM> view wave -undock
```

図7のWaveウィンドウが表示されます．Waveウィンドウの左側には追加された信号名がフルパスで表記されています．本ツールでは，スラッシュ(/)を用いた階層名表記を用いています．例えば最上位階層の`clk`信号は，「/top_pm/clk」と表示されます．

なお，GUIの操作や機能は，使用するシミュレータやデバッガに大きく依存します．ほとんどのツールが同様の機能を備えていますが，その操作方法や見えかたはツールによって異なります．詳細については，各ツールのマニュアルを参照してください．

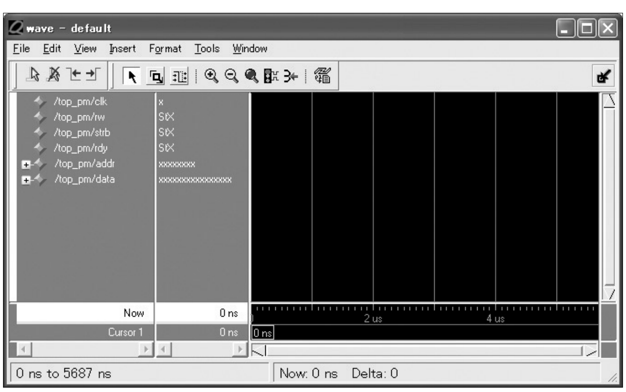


図7　Waveウィンドウ
Waveウィンドウは，波形を観測するためのデバッガである．

```
#     20: Starting Read/Write test
#     20: Writing data=298a to addr=71
#    220: Writing data=5672 to addr=9
              ⋮
             中略
              ⋮
#   3740: Reading data=55aa from addr=45
#   3980: Reading data=0fff from addr=39
# Read/Write test done
# ** Note: $stop    : proc.sv(87)
#    Time: 3980 ns  Iteration: 1  Instance: /top_pm/p
```

図8　シミュレーションのメッセージ（第1ステップ）
シミュレーションが`$stop`タスクにより3980nsで停止したことを示す．

● シミュレーションを実行する

観測信号を追加してシミュレーションの準備が整いました．実際にシミュレーションを実行してみましょう．コマンド・ラインから次のコマンドを入力します．

リスト1　プロセッサ内のテストベンチ記述

```
task read;                                          ← リード・タスク
    input  ['addr_size-1:0] a;
    logic ['word_size-1:0] d;
    begin
        addr_r = a;  }
        rw_r = 1;    }  ← 制御信号の出力
        strb_r = 0;  }
        @(posedge clk) strb_r = 1;
        while (rdy != 0) @(posedge clk) ;           ← rdy信号のウェイト
        d = data;
        $display("%t: Reading data=%h from
                        addr=%d", $time, d, a);     ← メッセージ表示
    end
endtask
task write;                                         ← ライト・タスク
    input  ['addr_size-1:0] a;
    input  ['word_size-1:0] d;
    begin
        $display("%t: Writing data=%h to
                        addr=%d", $time, d, a);     ← メッセージ表示
        addr_r = a;  }
        rw_r = 0;    }  ← 制御信号の出力
        strb_r = 0;  }
        data_r = d;  }
        @(posedge clk) strb_r = 1;
        while (rdy != 0) @(posedge clk) ;           ← rdy信号のウェイト
        data_r = 'bz;
        @(posedge clk);
    end
endtask
      <中略>
    write( 71, 16'h298A);  }
    write(  9, 16'h5672);  }
    write( 80, 16'hEFAC);  }
    write(135, 16'hAB00);  }
    write( 39, 16'h0FFF);  }  ← ライト・テスト
    write( 45, 16'h55AA);  }
    write(199, 16'hF197);  }
    write(125, 16'h0101);  }
    write(231, 16'h8954);  }

    read (  9);  }
    read ( 80);  }
    read (135);  }
    read ( 71);  }
    read (200);  }  ← リード・テスト
    read (125);  }
    read (231);  }
    read ( 45);  }
    read ( 39);  }
```

```
VSIM> run -all
```

このコマンドで，シミュレーション・イベントがすべて終了するまでシミュレーションが実行されます．コマンド・ラインに**図8**のようなメッセージが表示され，シミュレーションが`$stop`タスクにより3980ns（ナノ秒）で停止したことを示します．同時に，Waveウィンドウに波形が表示されます．

この動作はモジュール`proc`に記述されたパターン発生記述（**リスト1**）によるものです．このプロセッサ・モジュール内の記述がテストベンチということになります．このテストベンチは，二つのタスク`read`と`write`で構成されています．`read`は，アドレスを引き数にしてメモリからデータを読み出します．`write`は，アドレスとデータを引き数としてメモリにデータを書き込みます．ここでのアドレスは10進値で，データは16進値で指定しています．それぞれのタスクでは，`$display`タスクを用いて動作メッセージを出力します．

● 波形で動作を確認する

Waveウィンドウを用いて結果を確認します．操作方法の概略は**図9**に示します．

最初に基数の設定を行います．アドレスとデータの各信号名の上でマウスを右クリックして，ポップアップ・メニューから次の指定を行います．

- アドレス（/top_pm/addr）の基数を「Unsigned」に変更
- データ（/top_pm/data）の基数を「Hexadecimal」に変更

次に観測したい領域をズーミングします．まず，Zoom Fullアイコンをクリックして，波形全体を表示します．その後，フォーカスしたい領域をマウス中ボタン（またはスクロール・ボタン）のドラッグで選択します．波形のズームとスクロールを用いて，次の時間帯を表示します．同時

- 信号の移動：信号名をドラッグ・アンド・ドロップ
- 基数の変更：信号名で右クリックしたあと，図のメニューを選択
- 指定範囲ズーム：マウス中ボタンで領域をドラッグ
- 全体ズーム：Zoom Fullアイコンをクリック
- カーソル追加：Add Cursorアイコンをクリック
- 時間計測：複数のカーソルを計測点に配置
- カーソル配置：エッジの近くでカーソルをドロップ

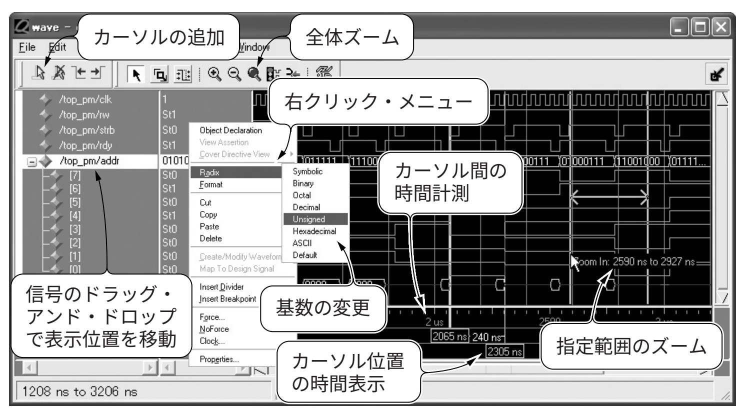


図9　Wave ウィンドウの使用方法

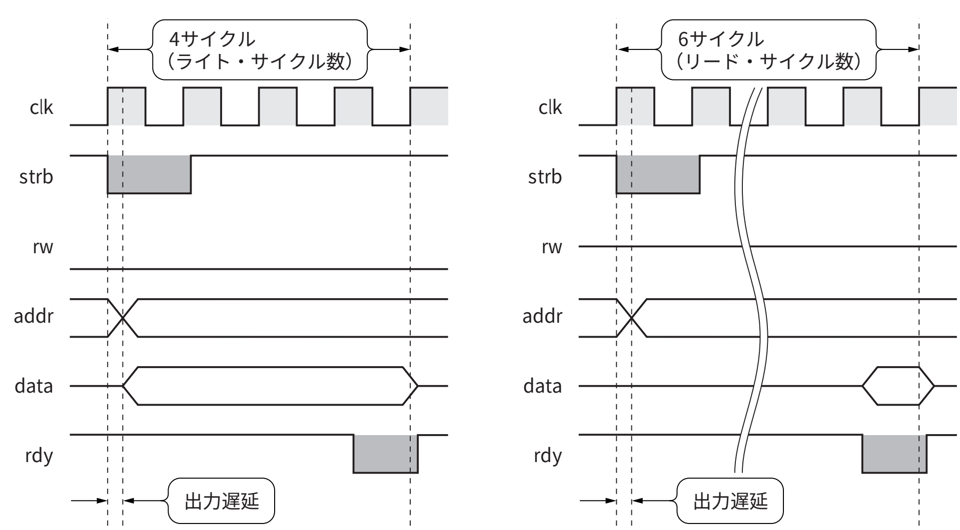


**図10
リード/ライト・サイクル**
このような図をタイミングチャートと呼ぶ．リード・サイクルはライト・サイクルと比べて長い．

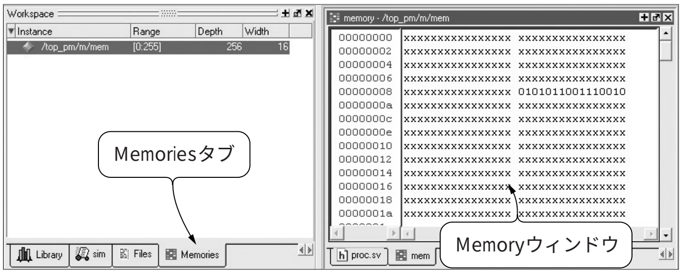


図11　設計データ内のメモリ
Memories タブ内では，設計データの中のメモリ配列を自動認識し，そのサイズやインデックス範囲を表示する．

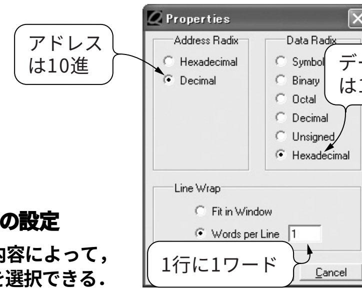


図12
メモリ表示方法の設定
メモリの構成や内容によって，自由な表示方法を選択できる．

にカーソルを信号の変化点（エッジ）に配置することで，その時間を確認します．

1）ライト・サイクル（220ns～385ns，図10（a））

プロセッサが，strb信号を1サイクルの間‘0’にします．このとき，クロックのエッジから5nsの遅延後にstrb信号が動いていることに注意してください．各モジュールの出力信号にはすべて，一定の遅延時間を設定してあります．これは，クロックと各信号が同時変化することを避けて，動作を見やすくするためです．

プロセッサはstrb信号と同時に，リード・ライト信号（rw）とアドレス値（addr），データ（data）を出力します．ライト・サイクル時はrw=0となります．メモリ側では，strb信号の入力を受けて，rw信号によりリード・ライトの判断を行います．strb信号から2サイクル後にメモリはデータを書き込んで，rdy信号をアクティブにします．次のクロック・エッジでプロセッサ側はrdy信号を検出してデータの出力を終了します．ライト・サイクルは，4サイクルで終了することになります．

2）リード・サイクル（2300ns～2545ns，図10（b））

プロセッサは，strb信号を1サイクル分アクティブにします．同時に，rw=1とアドレス値を出力します．メモリ側では，strb信号とrw信号の値によって4サイクル後にデータを出力し，rdy信号を返します．プロセッサは次のクロック・エッジでrdy信号を検出してデータの値を読み込みます．リード・サイクルは，計6サイクルになります．

● メモリの内容を確認する

本ツールには，Memoryウィンドウと呼ばれるデバッグ機能があります．このウィンドウでは，設計データ内に存在するメモリ配列を自動検出して，「表示」，「基数の変更」，「ファイルへの保存」，「ファイルの読み込み」，「メモリ内容の更新」などを行うことが可能です．

メモリ・ウィンドウを使用するには，WorkspaceのMemoriesタブをクリックします．このタブでは，/top_pm/m/memがすでにメモリとして認識されています（**図11**の左側）．ここでの表記は，Widthが16，Depthが256と表記されています．これは，「1ワードが16ビットで256個のアドレスを持つメモリ」であることを表しています．Rangeは実際のアドレス値の範囲を示しています．

メモリ名をダブルクリックすると，メイン・ウィンドウの右側にメモリの内容が表示されます（**図11**の右側）．表示されたMemoryウィンドウをアンドックします．メモリウィンドウのメニューから，「View」→「Properties」を選択します．表示されるダイアログ・ボックス（**図12**）で次のように設定します．

- Address Radix：Decimal
- Data Radix：Hexadecimal
- Line Wrap：Words per Line 1

この指定で，10進表記のアドレスの右に16進表記のデータが1ワードずつ表示されます．表示されるほとんどのアドレスは値が代入されていないため，「xxxx」となっています．テストベンチで代入したアドレス値を参考に，実際にメモリに値が書き込まれている状態を確認してください．

ここまでで，波形を用いたプロセッサとメモリのやりとり，およびMemoryウィンドウの操作方法を理解できたかと思います．

次のステップに進む前にいったんシミュレーションのセッションを終了します．コマンド・ラインに次のコマンドを入力します．

```
VSIM> quit -sim
```

これで，プロンプトが「QuestaSim>」に戻ります．シミ

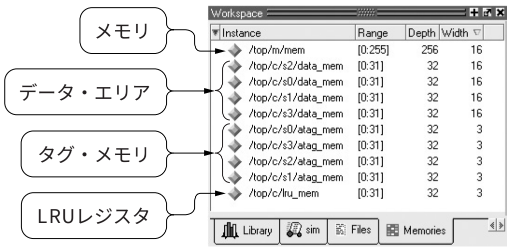


図13　キャッシュ内のメモリ構成
s0～s3の4セットのキャッシュ・メモリを確認できる．LRUレジスタは四つのキャッシュ・セットの使用履歴を保存する．

ュレータそのものを終了する場合は，

```
VSIM> quit -force
```

と入力します．

● **キャッシュを挿入する**

第2ステップでは，キャッシュを含んだシミュレーションを実行します．コマンド・ラインに次のコマンドを入力します．

```
QuestaSim> vsim top
```

ローディングされた設計データを，Workspaceのツリーで確認します．第1ステップの設計データに加えてモジュールcacheがcという名まえでインスタンスされています．また，モジュールcache内には四つのcache_setというモジュールがs0～s3という名称でインスタンスされています．これらが後述する四つのキャッシュ・セットになります．

Memoriesタブを選択して設計の中のメモリを確認します(**図13**)．第1ステップと同じように256アドレスを持つメモリを確認できます．キャッシュはs0～s3の4セットあり，それぞれのキャッシュ内にはメモリ・データを保存する16ビット×32アドレスで構成されたdata_memと，3ビット×32アドレスのatag_memの2種類のメモリが含まれています．さらに，lru_memという3ビット×32アドレスのメモリも確認できます．

図14のように，キャッシュの32アドレスで256アドレスのデータを扱うには，キャッシュの一つのアドレスにつき，8個のメモリ・アドレスを管理する必要があります．キャッシュは，メモリの内容を格納する領域以外にタグ・メモリという領域を持っています．atag_memがタグ・メモリに該当します．キャッシュにデータが書き込まれる際には，メモリ・アドレスの下位5ビットをエントリ・アドレスとしてタグ・メモリ上のアドレスを決めます．タグ・メモリには，メモリ・アドレス上位3ビットの値(フレーム・アドレス)が格納されます．例えばメモリ・アドレス135のデータが“FFFF”の場合を考えます．アドレス値を2進表記すると“10000111”となります．このうちの下位5ビットは“00111”で，10進数では7，上位3ビットは“100”です．このメモリ・アドレスのデータをキャッシュに格納する場合，キャッシュのアドレス7には“FFFF”を，タグ・メモリのアドレス7には“100”を格納します．同一のキャッシュ・アドレスは，同一のエントリ・アドレスを持つ八つのアドレス7，39，71，103，135，167，199，231によって共有されます．

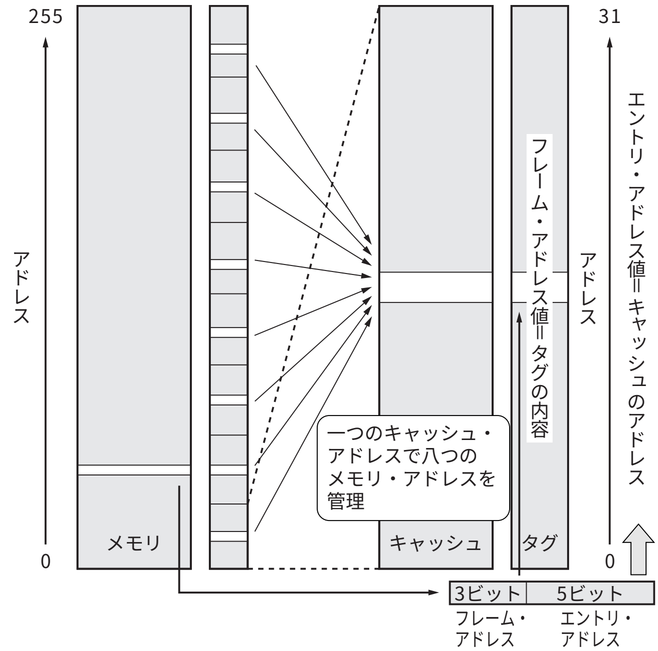


図14　キャッシュとメモリ
キャッシュでは，タグ・メモリを用いて保存しているアドレスの情報を管理している．タグ・メモリ内を検索することで，キャッシュ・ヒットとキャッシュ・ミスが判定される．

キャッシュ検索時には，まず使用するメモリ・アドレスからエントリ・アドレスを抽出してタグ・メモリの位置を検索し，その内容がフレーム・アドレスの値とマッチした場合にキャッシュ・ヒットとなります．

● **疑似LRUアルゴリズムに従ってキャッシュを書き換え**

このサンプル設計では，四つのキャッシュ・セットがあります．つまり同一のエントリ・アドレスに対して四つの

```
#    20: Starting Read/Write test
#    20: Writing data=298a to addr=71
#    60: Write miss, picking set 3
#   260: Writing data=5672 to addr=9
                     ⋮
                    中略
                     ⋮
#  3140: Reading data=8954 from addr=231
#  3180: Read hit to set 3
#  3220: Reading data=55aa from addr=45
#  3260: Read miss, picking set 0
#  3500: Reading data=0fff from addr=39
# Read/Write test done
# ** Note: $stop    : proc.sv(87)
#    Time: 3500 ns  Iteration: 1  Instance: /top/p
```

図15　シミュレーションのメッセージ（第2ステップ）
第1ステップのメッセージに加えて，キャッシュのヒット・ミスのメッセージが表示される．また，第1ステップよりシミュレーションの終了時間が早くなっている．

データ格納領域があることになります．同一のエントリ・アドレスに対するアクセスにおいて，異なったアドレスの参照が4回を超えた時点で，いずれかのキャッシュの内容を書き換える必要がでてきます．このとき利用されるのがLRU（least recently used）アルゴリズムです．

各キャッシュ・セットの中でもっとも使用されていないデータを検出します．このサンプル設計では，疑似LRUと呼ばれるある程度高い確率で使用履歴の少ないキャッシュ・セットを選択する方法を用いています．疑似LRUでは，四つのキャッシュ・セットに対して3ビットのレジスタで実装が可能です．lru_memというメモリがLRUの情報を保持するレジスタです．

● キャッシュを含めてシミュレーションを実行する

実際にシミュレーションを実行してみましょう．プロセッサ・モジュールは第1ステップと同一ですから，発生する信号パターンもまったく同じです．次のコマンドを実行してWaveウィンドウに観測信号を追加します．

```
VSIM> add wave /*top/*
```

第1ステップでは，strbやrw，rdyなどの信号で接続されていました．キャッシュを挿入することで各信号はキャッシュを介して接続されることになります．プロセッサとキャッシュは，pstrbやprw，prdyのように先頭に「p」を付加した信号を用いてマッピングしています．またキャッシュとメモリの間は，mstrbやmrw，mrdyのように「m」を付加した信号を接続に使用しています．キャッシュは，プロセッサからの制御信号を受け取ってキャッシュ・ミスが起きた場合にメモリ側へ制御信号を渡します．また，キャッシュ・ヒット時には，メモリが返すrdy信号はキャッシュからプロセッサへ渡されます．第1ステップと同じ方法でシミュレーションを実行します．

```
VSIM> run -all
```

図15のようなメッセージが表示されます．

第1ステップのメッセージに加えて，キャッシュ・ミスのメッセージが表示されます．また，第1ステップよりシミュレーションの終了時間が早くなっています．これは，キャッシュ・ヒットによって通常のメモリ・アクセスよりも短いサイクルで処理が終了したことを表しています．

● ライト・スルー方式を採用

リード・ヒット時には，データはキャッシュからの読み出しだけですが，ライト時にはキャッシュへの書き込みとメモリへの書き込みを行う必要があります．この書き込みのタイミングによって，キャッシュはライト・バック方式とライト・スルー方式に分けられます．ライト・スルー方式では，キャッシュへの書き込みと同時にかならず外部メモリへも書き込みを行います．ライト・バック方式は，ある条件が成立したときのみメモリへの書き込みを行います．

このサンプル設計ではライト・スルー方式を利用しています．キャッシュを挿入した場合も，メモリ・ライト時にはサイクル数は減りません．逆に，キャッシュ・ヒットの判断を行うために1サイクル長くなっています．このようなライト・スルー方式のキャッシュでは，ライト・バッファを挿入するのが一般的ですが，ここでは使用していません．ライト・バッファの挿入については，次回以降でトライします．

● シミュレーション結果を確認する

リード・ヒット時にはキャッシュの効果が現れ，サイクル数が減少します．このことを実際のシミュレーション波形で確認します．現在の波形と第1ステップの波形を並べて比較します．第1ステップの波形を読み込むには，次のコマンドを実行します．

```
VSIM> dataset open pm.wlf
```

これにより，第1ステップで保存しておいたpm.wlfが読み込まれ，Workspaceに「pm」というタブが追加されます．

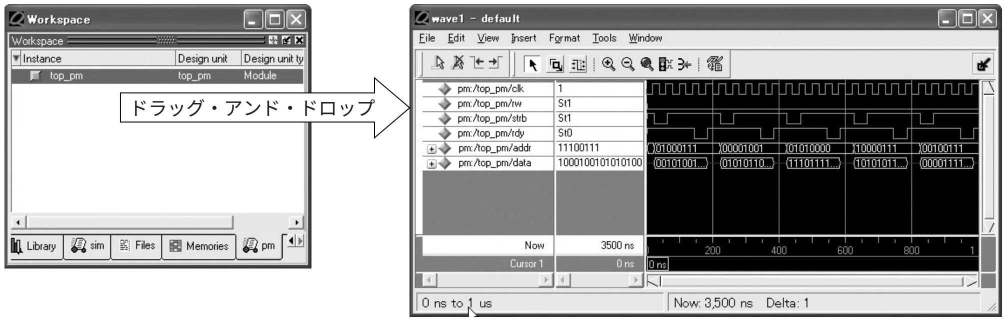


図16
ドラッグ・アンド・ドロップで信号を追加
階層名をドラッグ・アンド・ドロップすることで，その階層に含まれる信号をすべてWaveウィンドウに追加できる．

さらに，コマンド・ラインに次のコマンドを入力します．

```
VSIM> view wave -undock
VSIM> view -new wave -undock
```

Waveウィンドウがアンドックされて，さらに新しいWaveウィンドウが表示されます．**図16**のようにWork-spaceでpmタブを選択して，`top_pm`の階層を新しいWaveウィンドウにドラッグ・アンド・ドロップします．二つのWaveウィンドウで，それぞれアドレスとデータの基数を第1ステップと同様に変更します．**図17**のように，それぞれ次の時間帯を表示して，リード・ヒット時の動作の違いを確認します．

- 第1ステップ：1820ns～2540ns
- 第2ステップ：2180ns～2420ns

次にシミュレーション時のメッセージを確認します．

```
#    20: Writing data=298a to addr= 71
#    60: Write miss, picking set 3
```

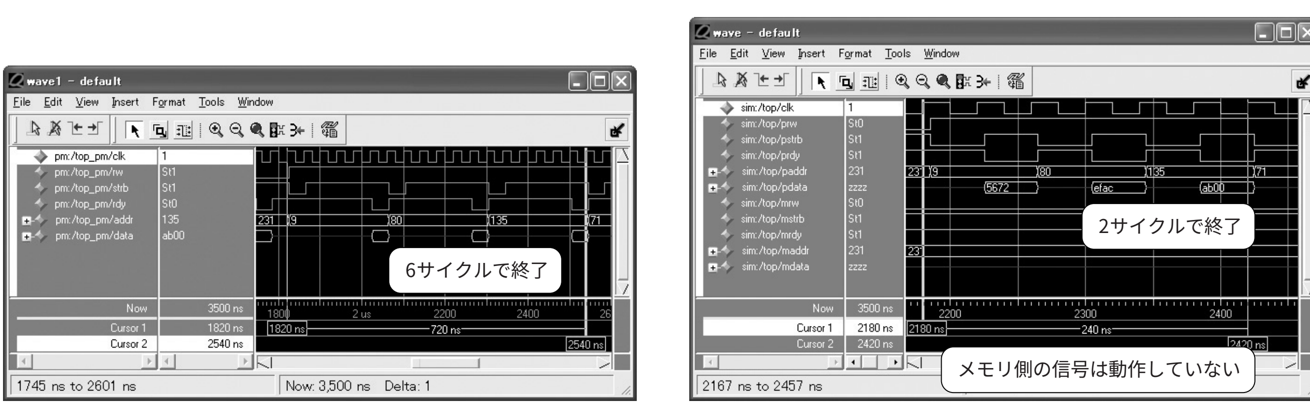
この2行はライト・サイクルを表しています．20nsでアドレス71にデータの書き込みを行っています．次のサイクルの60nsでキャッシュ・ミスしています．このときデータを書き込むキャッシュ・セットは，`s3`が選択されています．また，リード・サイクルの場合は，次のようになります．

```
#  3180: Read hit to set 0
#  3220: Reading data=55aa from addr= 45
```

リード・サイクルでは，データをプロセッサが読み込んだ時点でメッセージを表示するため，キャッシュのヒット・ミスのメッセージが先に表示されます．ここでは，アドレス45に対する読み込みでキャッシュ・ヒットしています．ヒットしたキャッシュ・セットは`s0`になります．

メッセージを確認することで，次の検証項目を確認できます．同一のエントリ・アドレスを持つ71，135，39，199，231のアドレスへ順番に書き込みを行って，アドレス71のデータが上書きされています．その後のアドレス135，71に対するリードでは，135でヒット，71でミスしています．メッセージからこの情報を抽出して下記に示します．

- 20ns：addr＝71に書き込み → 60ns：ライト・ミス，キャッシュ・セット`s3`を選択
- 740ns：addr＝135に書き込み → 780ns：ライト・ミ



（a）第1ステップの結果　（b）第2ステップの結果

図17　リード・ヒット時の効果を確認
キャッシュを挿入した場合は，`strb`信号を検出したサイクルで`rdy`信号が出力され，キャッシュからデータの読み出しが行われる．

ス，キャッシュ・セット`s1`を選択
- 980ns：addr＝39に書き込み → 1020ns：ライト・ミス，キャッシュ・セット`s2`を選択
- 1460ns：addr＝199に書き込み → 1500ns：ライト・ミス，キャッシュ・セット`s0`を選択
- 1940ns：addr＝231に書き込み → 1980ns：ライト・ミス，キャッシュ・セット`s3`を選択

最後の書き込みで，アドレス71のデータ（キャッシュ・セット`s3`）をアドレス231のデータで上書きしています．ここでは，もっとも古く利用されたデータが選択されています．さらに，

- 2380ns：キャッシュ・ヒット → 2420ns：アドレス135からのリード
- 2460ns：キャッシュ・ミス → 2700ns：アドレス71からのリード

このようにして，ほかのリード・ライトについても動作を確認します．

● 信号パターンを編集して再シミュレーション

第3ステップでは，プロセッサ・モジュール内の`read/write`のタスクを使用して信号パターンを書き換えます．ここでのポイントは，各アドレスのエントリ・アドレスや使用されるキャッシュ・セットを考慮してパターンを構成することです．第2ステップのシミュレーションでその一例を確認しました．

信号パターンの修正とシミュレーションを繰り返すためには，シミュレーション・セッションを終了せずに，「ソースの修正」，「再コンパイル」，「リスタート」を繰り返します．まず，Workspace上でモジュール`proc`をダブルクリックします．Sourceウィンドウにソース・コードが表示されます．ソース・コード上で右クリック・メニューから「Read Only」のチェックを外します．その後，`read/write`のタスクの構成や引き数を編集します．編集して保存しない状態では，Sourceウィンドウのproc.svタブには「*」マークが付いています．編集後，Saveアイコン をクリックして上書き保存します．このとき，「*」マークが消えることを確認してください．次に，コマンド・ラインからコンパイルを実行します．

```
VSIM> vlog -incr *.sv
```

「-incr」は，インクリメンタル・コンパイル・オプションで，変更のあったデータ（proc.sv）のみをコンパイルします．さらにコマンド・ラインからリスタートを実行します．

```
VSIM> restart -force
```

このコマンドで，シミュレータは時間0に戻り，変更された設計データのみ再ローディングします．ここでは，プロセッサだけがローディングされます．また，Waveウィンドウにリストされた観測信号は維持されているので，そのままrunを実行するだけで再シミュレーションが可能です．

信号パターンは，現在記述されている`write/read`タスクの並べ替えや，引き数のアドレスとデータの変更，タスクの追加，`repeat`や`for`ループを使用した繰り返し処理などを入れてみてください．

再シミュレーションでは，出力されるメッセージによってキャッシュ・ヒット/ミスの状況を確認するとともに，WaveウィンドウとMemoryウィンドウを使用して外部メモリと四つのキャッシュ・セットの内容，およびタグ・メモリの内容などを確認してください．LRUレジスタの動作を確認するのもおもしろいと思います．

● 次回はパターン生成自動化と達成度の計測を解説

ここまで実行してきて，パターンの入力にもう少し効率の良い方法はないものか，という感想を持たれた方がいるかもしれません．本連載の目的の一つは，このようにあるルールを持ったテストベンチを自動的に生成して検証の効率を上げる方法を説明することにあります．次回は，同じサンプル設計を用いて信号パターンの発生を自動化する方法と，自動テストで検証項目の達成度を計測する方法を紹介します．

もりた・えいいち
メンター・グラフィックス・ジャパン（株）

<筆者プロフィール>
森田栄一．Questa（旧名V-System）にかかわり始めたころは20代だったのに，気づけば40代間近．HDL設計環境を見続けてきて感じるのは，その進化の遅さ．設計規模と速度は2けたも向上しているのに，検証手法はやっとSystemVerilogなどが登場してきたにすぎない．そんなに速くてもついていけないのかもしれないが…．